

Snapping Graph Drawings to the Grid Optimally

Andre Löffler, Thomas C. van Dijk, and Alexander Wolff

Lehrstuhl für Informatik I, Universität Würzburg, Germany.
<http://www1.informatik.uni-wuerzburg.de/en/staff>

Abstract. In geographic information systems and in the production of digital maps for small devices with restricted computational resources one often wants to round coordinates to a rougher grid. This removes unnecessary detail and reduces space consumption as well as computation time. This process is called *snapping to the grid* and has been investigated thoroughly from a computational-geometry perspective. In this paper we investigate the same problem for given drawings of planar graphs under the restriction that their combinatorial embedding must be kept and edges are drawn straight-line. We show that the problem is NP-hard for several objectives and provide an integer linear programming formulation. Given a plane graph G and a positive integer w , our ILP can also be used to draw G straight-line on a grid of width w and minimum height (if possible).

1 Introduction

When compressing geographic data, for example in order to ship it to devices with small memory, small screens and slow CPUs, the main objective is to reduce unnecessary detail. One way to do this is to round data points to a grid.

In the computational geometry community, a process called *snap rounding* has been proposed and has since become well-established: given an arrangement of line segments, each grid cell that contains vertices or intersections is “hot”. Then every segment becomes a polygonal chain whose edges (*fragments*) connect center points of hot cells, namely those that the original segment (*ursegment*) intersects. Guibas and Marimont [7] showed that during snap rounding, vertices of the arrangement never cross a polygonal chain, so after snapping no two fragments cross. Moreover, the circular order of the fragments around an output vertex is the same as the order in which the corresponding ursegments intersect the boundary of its grid cell. The resulting arrangement approximates the original one in the sense that any fragment lies within the Minkowski sum of the corresponding ursegments and a unit square centered at the origin. However, the structure of the graph can be affected (vertices merge, faces disappear, edges bend). Further work in this direction includes that of De Berg et al. [3].

Motivated by the above GIS application, we investigate the problem of moving the drawing of a graph to a given grid. Since we still want to be able to recognize the original graph, we do not tolerate new incidences. Then we must accept the possibility that a vertex does not go to the nearest grid point, but we

still want to minimize change. This can be measured, for example, by the sum of the distances or the maximum distance in the Euclidean (L_2 -) or Manhattan (L_1 -) metric. Apparently, this problem, which we call **TOPOLOGICALLY-SAFE SNAPPING**, has not been studied yet. (Note that we carry over the term “snapping,” although we don’t necessarily snap to the *nearest* grid point.)

From a graph-drawing perspective, restricting to the grid has a (relatively) long history. Motivated by the fact that Tutte’s barycenter method [14] for drawing planar graphs yields drawings that need precision linear in the size of the graph, Schnyder [13] and, independently, de Fraysseix et al. [5] have shown that any planar graph with n vertices admits a straight-line drawing on a grid of size $O(n) \times O(n)$. This is asymptotically optimal in the worst case [5]. Chrobak and Nakano [2] have investigated drawing planar graphs on grids of smaller width, at the expense of a larger height. Grid-snapping techniques can be found in any diagram creation tool. Aesthetic properties of force-directed drawing algorithms are widely researched, see e.g. Kieffer et al. [8] for grid layouts of diagrams.

Although minimizing the area of straight-line grid drawings has been the topic of several graph drawing contests, there has been rather little previous work. It is known that the problem is \mathcal{NP} -hard [9], but not even for special cases exact or approximation algorithms have been proposed.

Our contribution. We show that optimal snapping is \mathcal{NP} -hard, with a reduction that asks for compressing each coordinate by just a single bit (Sect. 2). The proof is somewhat similar in concept to the proof of the \mathcal{NP} -hardness of Metro-Map Layout [11,15], but new constructions are required since the snapping problem does not easily allow the construction of “rigid” gadgets. Second, we give an integer linear program (ILP) for optimal snapping (Sect. 3). This ILP generalizes the one for Metro-Map Layout [12]. Where that ILP assumes a constant number of possible edge directions (namely 8), we have to cope with a number that is quadratic in the size of the grid. The numbers of variables and constraints of our ILP are polynomial in grid and graph size, but are quite large in practice. In fact, on a grid of size $k \times k$, there are $\Theta(k^2)$ edge directions. Thus, for an n -vertex planar graph, we must generate $O(k^2 n^2)$ constraints, among others, to preserve planarity and the cyclic order of edges around the vertices. To ameliorate this, we apply delayed constraint generation, a technique that adds certain constraints only when needed. Still, runtime is prohibitive for graphs with more than about 15 vertices. Our techniques can be adapted to draw (small) graphs with minimal area. This is interesting even for small graphs since minimum-area drawings can be useful for validating (counter)examples in graph drawing theory.

2 NP-Hardness

We start with a formal definition of **TOPOLOGICALLYSAFE SNAPPING** – or **TSS** for short. To measure the cost of rounding a graph, we utilize Manhattan distance and the total cost of rounding a graph is the sum over the individual costs of the vertices. As input we take a plane graph $G = (V, E)$ with vertex positions

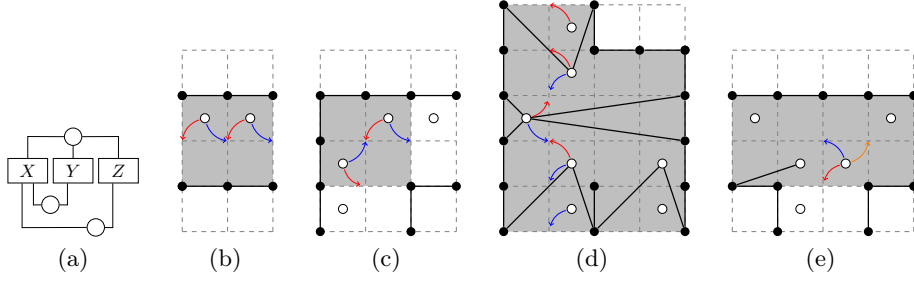


Fig. 1. (a) Graph $H(F)$ for formula $F = (\overline{X} \vee \overline{Y} \vee \overline{Z}) \wedge (X \vee Y) \wedge (X \vee Z)$, (b) horizontal line gadget, (c) bottom-to-right bend gadget, both with possible roundings, (d) gadget for variable with two negated and one unnegated occurrences, (e) all-negated clause gadget with three negated variables. Inner area of each gadget highlighted gray.

and a bounding box $[0, X_{\max}] \times [0, Y_{\max}]$. The TSS problem is then to minimize the cost of rounding the vertices of G to the integer grid within the box without altering the topology with respect to the given plane straight-line drawing of G .

We prove \mathcal{NP} -hardness of TSS by considering the decision variant: is there a rounding that does not exceed a given cost bound c ? We reduce from PLANAR MONOTONE 3-SAT (which is \mathcal{NP} -hard [4]): given a formula F in 3-CNF that is monotone and whose graph $H(F)$ is planar, is F satisfiable? The graph $H(F)$ has a vertex for each variable and each clause of F and an edge between a variable vertex v_X and a clause vertex v_C if X is part of C . We will only consider formulae whose graphs are planar and that are *monotone* in the usual sense: for any clause C , variables in C either are all negated or all unnegated. We can assume that the graph $H(F)$ can be laid out as in Fig. 1 (a): all variable vertices lie on the x-axis, the vertices of all-negated clauses lie above the x-axis, and the vertices of all-unnegated clauses lie below the x-axis [4].

Theorem 1. TOPOLOGICALLYSAFESNAPPING is \mathcal{NP} -hard.

Proof. For a given monotone, planar 3-CNF formula F , we construct a cost bound c_{\min} and a plane graph G with vertices at half-integer coordinates. The sum of all vertex movements induced by rounding G to integer coordinates is exactly c_{\min} if and only if F is satisfiable. To achieve this, we introduce gadgets for the elements of $H(F)$ – variables, clauses, edges and bends – and construct G and c_{\min} in polynomial time.

For exposition, we consider two types of vertices. Black vertices start on integer grid points and do not need to be rounded. Moving a black vertex to another integer grid point is allowed, but we will show that this is not optimal if F is satisfiable. White vertices start at grid cell centers and thus will always move at least one unit by rounding. Let $W \subseteq V(G)$ be the set of white vertices. Now we give the construction of the various gadgets.

First, we introduce the line and bend gadgets. These ensure consistency between variable and clause gadgets. Every segment of the line gadget consists of

four black vertices and two edges forming a *tunnel*, and a single white vertex inside; see Fig. 1 (b). The white vertex can be rounded most cheaply to exactly two possible integer grid points, depicted by the red and blue arrows. By rounding a white vertex in one direction, we prohibit the neighbor in that direction to go the opposite way – as both vertices would end up on the same integer grid point (which violates topological safety). So, if the white vertex at one end of the line is rounded inward (blue arrow) the white vertex at the other end of that line must be rounded outward – we say it is *pushed*. The same holds for the bend gadgets, as can be seen in Fig. 1 (c).

Next, consider the variable gadget depicted in Fig. 1 (d). It has tunnels for vertical line gadgets for every negated and unnegated occurrence at the top and bottom respectively. At the center of this gadget, there is a white vertex that is connected to the gadget’s walls by two triangles. Call this the *assignment* vertex and note that it can be rounded up or down, which makes the edges of the triangles block grid points on the top or bottom tunnels, respectively. The tunnels of that direction are then all forced to push into the connected clause gadgets. This represents the truth assignment of the corresponding variable.

Finally, the clause gadget is shown in Fig. 1 (e). We describe the all-negated degree-3 version; the degree-2 version can be constructed similarly. There is a white *satisfaction* vertex that can go to any of three possible integer grid points at equal cost. These grid points belong to line gadgets and are only available if the line does not “push.” Then the satisfaction vertex can be rounded at cost 1 if and only if the clause is satisfied. Gadgets for all-unnegated clauses can be obtained by mirroring the construction of Fig. 1 (e) at a horizontal line.

The rounding cost of G is bounded from below by $c_{\min} = |W|$ since every white vertex must be rounded at cost at least 1. If F is satisfiable, there is a rounding that achieves this because then we can round the assignment vertices such that the satisfaction vertices can be rounded at cost 1. In the other direction, a satisfying assignment can be read off from the assignment vertices if rounding occurred at cost c_{\min} .

If none of the three candidate grid points for the satisfaction vertex are available, a topologically correct rounding must move a black vertex associated with that clause (of either the clause itself, the connected variables or the edges and bends connecting them). This adds at least 1 to the rounding cost without reducing the movement of any white vertex and thus such solutions cost strictly more than c_{\min} . That is, if c_{\min} is exceeded, then F is unsatisfiable: any rounding corresponding to a satisfying truth assignment is cheaper. This concludes our Karp reduction and the claim follows. \square

Corollary 1. *TOPOLOGICALLYSAFE SNAPPING is also \mathcal{NP} -hard when using Euclidean distance. In this case it is also \mathcal{NP} -hard to minimize the maximum movement instead of the sum.*

Proof (sketch). The above proof goes through with Euclidean distance and $c_{\min} = \sqrt{0.5^2 + 0.5^2} \cdot |W|$. For minimizing the maximum movement, observe that rounding white vertices now costs less, but moving a black vertex still has cost at

least 1: if F is satisfiable, the maximum movement is $\sqrt{0.5^2 + 0.5^2}$, otherwise it is at least 1. \square

This distinction of maximum movement ($\sqrt{0.5^2 + 0.5^2} \approx 0.71$ versus 1) based on the satisfiability of F also gives the following.

Corollary 2. *Euclidean TOPOLOGICALLYSAFE SNAPPING with the objective to minimize maximum movement is \mathcal{APX} -hard.*

3 Exact Solution using Integer Linear Programming

In this section we provide an ILP-based exact algorithm for TSS. Recall that an instance is a graph $G = (V, E)$ with vertex coordinates. For all $v \in V$, call these (X_v, Y_v) and introduce integer decision variables $0 \leq x_v \leq X_{\max}$ and $0 \leq y_v \leq Y_{\max}$ to represent the “rounded” output position. This leads to the following objective function.

$$\text{Minimize } \sum_{v \in V} |x_v - X_v| + |y_v - Y_v| \quad (1)$$

This formula is itself not linear, but can be made so with standard transformations [10]. Note that without any further constraints, this would just move every vertex to the nearest integer grid point. We will now introduce constraints to ensure topological safety, that is, in the output no two points are on same grid point, no two edges intersect, and the edges at every node have the same cyclic order as in the input.

Vertices do not coincide. This can be ensured by adding the following constraints. They too are not linear as stated, but can be readily linearized.

$$(x_v \neq x_w) \vee (y_v \neq y_w) \quad \forall v, w \in V, v \neq w \quad (2)$$

Possible directions. The most important departure from the metro-map drawing ILP is that, clearly, more than eight different directions are allowed. A priori we have no further constraints than that every rounded vertex lies somewhere within the given bounding box. Let \mathcal{D} be the set of unique directions $D = (D_X, D_Y)$ in $[-X_{\max}, X_{\max}] \times [-Y_{\max}, Y_{\max}]$. Considering the Farey sequence [6], we know that $|\mathcal{D}|$ is $\Theta(X_{\max} \cdot Y_{\max})$. In the following, we let the set \mathcal{D} be ordered counterclockwise, starting at the positive x-axis, allowing comparison of directions.

No two edges cross. The following constraints ensure that nonincident edges do not cross. (Incident edges are allowed to touch in the shared vertex.) We will follow the idea of Nöllenburg and Wolff [12]. While producing octilinear drawings of metro maps, they ensured planarity by forcing every pair of nonincident edges to be separated by at least some distance D_{\min} in at least one of the eight octilinear directions. This minimum distance was partly an aesthetic guideline, but also guarantees planarity. We are only interested in the latter and therefore pick D_{\min} such that all planar realizations on the grid are allowed.

The separation distance D_{\min} has to be small enough to separate any non-intersecting pair of edges in the output. Here the bounding box leads to a bound since it bounds the slope of the edges; it suffices to choose $D_{\min} = 1/(\max\{X_{\max}, Y_{\max}\} + 1)$.

For every pair of nonincident edges $e_1, e_2 \in E$ and every direction $D \in \mathcal{D}$, we introduce a binary decision variable $\gamma_D(e_1, e_2) \in \{0, 1\}$ indicating that e_1 and e_2 are apart by D_{\min} in direction D . Every such pair must be separated in some direction (following the idea of [11]).

$$\sum_{D \in \mathcal{D}} \gamma_D(e_1, e_2) = 1 \quad \forall e_1, e_2 \in E, e_1, e_2 \text{ nonincident} \quad (3)$$

Let $L_\gamma = 2 \cdot \max\{X_{\max}, Y_{\max}\} + 1$. Then, for any direction $D \in \mathcal{D}$, any pair of nonincident edges e_1, e_2 and any $v \in e_1, w \in e_2$, we require the following.

$$D_X \cdot (x_v - x_w) + D_Y \cdot (y_v - y_w) + (1 - \gamma_D(e_1, e_2))L_\gamma \geq D_{\min} \quad (4)$$

Constraint (3) yields a unique direction D with $\gamma_D = 1$. By choice of L_γ , any constraint (4) that involves a direction D with $\gamma_D = 0$ is trivially fulfilled.

Determine direction of incident edges. For incident edges $e_1, e_2 \in E$, we have to ensure that the directions of e_1 and e_2 differ. Again, we generalize the metro-map drawing ILP – dropping the “relative position rule” – allowing edges to have any direction $D \in \mathcal{D}$.

To keep track of this, we introduce a binary decision variable $\alpha_D(v, w) \in \{0, 1\}$ for every vertex $v \in V$, every neighbor $w \in N(v)$ and every direction $D \in \mathcal{D}$. The meaning of $\alpha_D(v, w) = 1$ is that the direction of edge (v, w) is D .

$$\sum_{D \in \mathcal{D}} \alpha_D(v, w) = 1 \quad \forall v \in V \forall w \in N(v) \quad (5)$$

For any vertex $v \in V$, any neighbor $w \in N(v)$, and any direction $D \in \mathcal{D}$, the following ensures that edge (v, w) indeed has direction D . Let $L_\alpha = 2 \cdot \max\{X_{\max}, Y_{\max}\} + 1$.

$$\begin{aligned} x_w \cdot D_Y + y_v \cdot D_X - x_v \cdot D_Y \pm (1 - \alpha_D(v, w))L_\alpha &\geq y_w \cdot D_X \\ (1 - \alpha_D(v, w))L_\alpha + (x_w - x_v) \cdot D_X + (y_w - y_v) \cdot D_Y &\geq 0 \end{aligned} \quad (6)$$

From constraint (5) we get that for every vertex-neighbor pair one α has to be set to 1. This α again enables one subset—as L_α dominates all other terms—of constraints from (6), forcing comparison between edge slope and direction. This gives us the direction of edge (v, w) with the correct sign.

Preserve cyclic order of outgoing edges. We use a binary decision variable $\beta(v, w) \in \{0, 1\}$ for every vertex-neighbor pair, indicating if w is the “last” neighbor of v according to the order of \mathcal{D} . The following preserves cyclic order.

$$\sum_{w \in N(v)} \beta(v, w) = 1 \quad \forall v \in V \text{ with } \deg(v) > 1 \quad (7)$$

$$\begin{aligned} \alpha_{D_1}(v, w_i) &\leq \beta(v, w_i) + \sum_{D_w \in \mathcal{D}: D_w > D_1} \alpha_{D_w}(v, w_{i+1}) \\ \forall D_1 \in \mathcal{D} \forall v \in V, N(v) &= \{w_1, w_2, \dots, w_k\} \ (k = \deg v > 1) \end{aligned} \quad (8)$$

For notational convenience, we let $w_{k+1} = w_1$, as $N(v)$ is conceptually circular. For any α set to 0, the inequalities of (8) are trivially satisfied. Otherwise, there has to be a neighbor whose connecting edge has a later direction (and thus the corresponding α set to 1), unless it is the last neighbor in the embedding of v . To ensure that there is only one “last neighbor”-violation of the constraints from (8), we introduce the constraints of (7). Adding β to every constraint of (8) also allows for the whole neighborhood of v to be rotated around it. This describes the full ILP and gives to the following.

Theorem 2. *The above ILP solves TOPOLOGICALLYSAFE\$SNAPPING.*

Graph drawing. Replacing the objective function with $\text{Minimize } \max_{v \in V} y_v$, the ILP computes a straight-line grid drawing with the given embedding, width at most X_{\max} , and minimum height. This allows us to find minimum-area drawings of small graphs.

Delayed constraint generation. We can apply a delayed constraint generation approach (see for example Cinneck [1]) to the above ILP as follows. First we run the ILP without any constraints, which snaps each vertex to the nearest grid point. (This takes practically no time.) We then test the result for topological validity, adding constraints corresponding to any violations. Then we repeat until no violations occur. This improves the runtime when few iterations suffice for a particular instance, but the approach should still be considered practically infeasible, especially for large bounding boxes: the set of possible directions \mathcal{D} still results in a large program. Future work could focus on reducing the brute-force inclusion of all possible directions. Experimental results are found in the appendix.

Acknowledgments. We thank Gergely Mincsovics for suggesting this problem to us.

References

1. J. W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*. Springer, 2008.
2. M. Chrobak and S.-I. Nakano. Minimum-width grid drawings of plane graphs. *Comput. Geom.*, 11(1):29–54, 1998.
3. M. de Berg, D. Halperin, and M. Overmars. An intersection-sensitive algorithm for snap rounding. *Comput. Geom.*, 36(3):159–165, 2007.
4. M. de Berg and A. Khosravi. Optimal binary space partitions for segments in the plane. *Int. J. Comput. Geom. Appl.*, 22(03):187–205, 2012.
5. H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

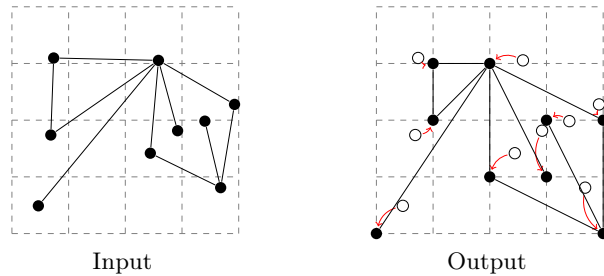
6. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics—A Foundation for Computer Science*. Addison-Wesley, Reading, MA, U.S.A., 1994.
7. L. J. Guibas and D. H. Marimont. Rounding arrangements dynamically. *Int. J. Comput. Geom. Appl.*, 8(2):157–178, 1998.
8. S. Kieffer, T. Dwyer, K. Marriott, and M. Wybrow. Incremental grid-like layout using soft and hard constraints. In S. K. Wismath and A. Wolff, editors, *Proc. 21st Int. Symp. Graph Drawing (GD’13)*, volume 8242 of *LNCS*, pages 448–459. Springer, 2013.
9. M. Krug and D. Wagner. Minimizing the area for planar straight-line grid drawings. In S.-H. Hong, T. Nishizeki, and W. Quan, editors, *Proc. 15th Int. Symp. Graph Drawing (GD’07)*, volume 4875 of *LNCS*, pages 207–212. Springer, 2008.
10. B. A. McCarl and T. H. Sreen. *Applied mathematical programming using algebraic systems*. Texas A&M University, 1997.
11. M. Nöllenburg. Automated drawing of metro maps. Master’s thesis, Fakultät für Informatik, Universität Karlsruhe, 2005. Available at <http://www.ubka.uni-karlsruhe.de/indexer-vvv/ira/2005/25>.
12. M. Nöllenburg and A. Wolff. Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE Trans. Visual. Comput. Graphics*, 17(5):626–641, 2011.
13. W. Schnyder. Embedding planar graphs on the grid. In *Proc. 1st ACM-SIAM Symp. Discrete Algorithms (SODA’90)*, pages 138–148, 1990.
14. W. T. Tutte. How to draw a graph. *Proc. London Math. Soc.*, 13(52):743–768, 1963.
15. A. Wolff. Drawing subway maps: A survey. *Informatik – Forschung & Entwicklung*, 22(1):23–44, 2007.

Appendix: Experimental evaluation

In the following section, we will discuss performance of an IBM CPLEX implementation of the above ILP on graphs different in vertex count, size of bounding rectangle and number of “difficult” parts. We ran experiments on a Linux machine with 16 cores (2666 MHz and 4 MB cache each), 16 GB memory and 20 GB swap space and using the Java bindings for Cplex. Model size is measured by considering the number of rows and columns before and after CPLEX completes any preprocessing step, “*” means that no model could be created within given time and memory. Any times given are in wall clock time, and “†” means that there was no result within 10 minutes of computation.

In the following, *full model* is used for executions of the above ILP without row generation. The column *first* gives the time until any feasible integer solution (not necessarily optimal) is reported by the integer solver. In both cases, *optimal* gives the time until the solver reports an optimal solution. For the output figures, white vertices represent the initial positions with the red arrows indicating actual vertex movement.

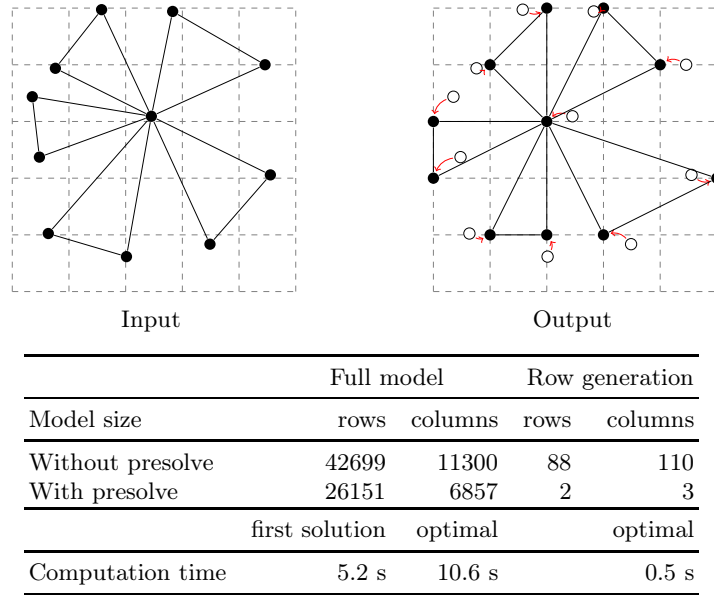
We start with a rather small graph (Fig. 2). Because of its size, building the model and finding the solution is quick. However, its vertices are positioned so that many constraints are not trivially satisfied and significant effort is required even by the row generation approach.



	Full model		Row generation	
Model size	rows	columns	rows	columns
Without presolve	12809	3631	3795	1133
With presolve	8046	2239	3791	1053
	first solution		optimal	optimal
Computation time	3.2 s	90.6 s		29.2 s

Fig. 2. Graph 1: $|V| = 9, |E| = 10$

In Fig. 3, the by far most expensive constraint is for checking the embedding of the central node. Any other constraint is easily satisfiable. In terms of computation time, there is not a big difference between finding the first solution and

**Fig. 3.** Graph 2: $|V| = 11$, $|E| = 15$

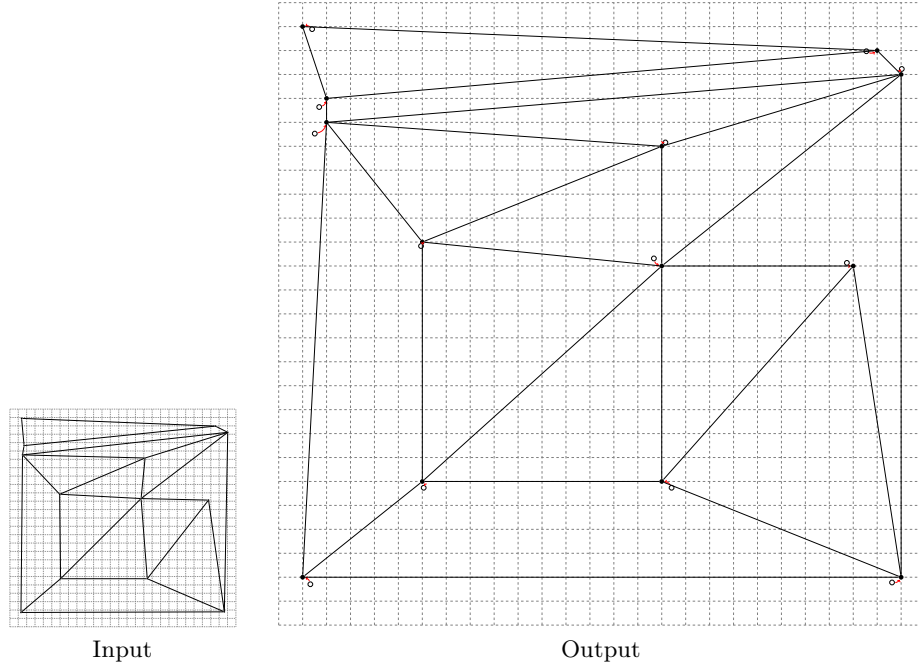
closing the integrality gap. Notice, that every vertex has one preferred integer grid point that is not preferred by any other vertex, so just rounding to the nearest grid point already gives a optimal solution. This solution is found by the first run of the row generation approach almost immediately. (Note that the 0.5 second runtime includes setting up the Java environment, calling the CPLEX solver and checking topology.) The difference between the full model and the row generation approach becomes even more important when the size of the bounding box increases. Consider Fig. 4. While still easy to round in the same sense as above, as size of the bounding rectangle increases, so does the time for building and solving the full model. However, this has no impact on solving time for the row generation approach for “easy” graphs.

Consider the graph in Fig. 5. While too large to round with the full model approach in the allotted time of 10 minutes, the row generation only has to add one constraint from Equation 2 for two vertices of the upper-right corner. Rebuilding the model and solving with this constraint runs in reasonable time (compared to the full model). Notice that this constraint does not involve the direction set \mathcal{D} .

When rounding graphs with vertices starting in close proximity (like in Fig. 6) several things can be noticed. First of all, small bounding box result in small and easy-to-solve models. The size of the bounding box has extreme effect on the runtime (compare Fig. 2, which has only two more vertices but a much larger bounding box). Second, when many constraints are violated during the

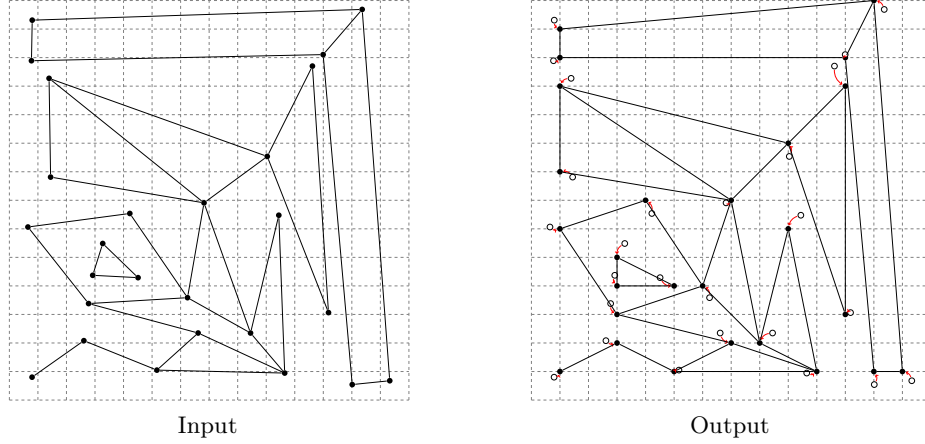
row generation processes, iteratively adding the constraints results in runtime exceeding the time for solving the full model in the first place.

We end this section with two rather small examples (Figs. 6 and 7). Both have a rather large number of vertices compared to the size of the bounding rectangle and thus include many “difficult” parts. The row generation approach clearly outperforms the full model (while still being infeasible in practice).

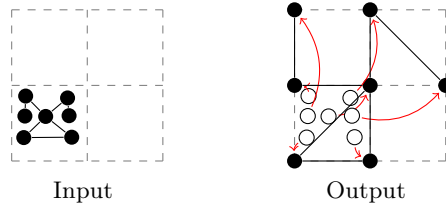


	Full model		Row generation	
	rows	columns	rows	columns
Model size				
Without presolve	*	*	104	130
With presolve	*	*	4	5
	first solution	optimal	optimal	
Computation time	†	†	0.4 s	

Fig. 4. Graph 3: $|V| = 13$, $|E| = 25$

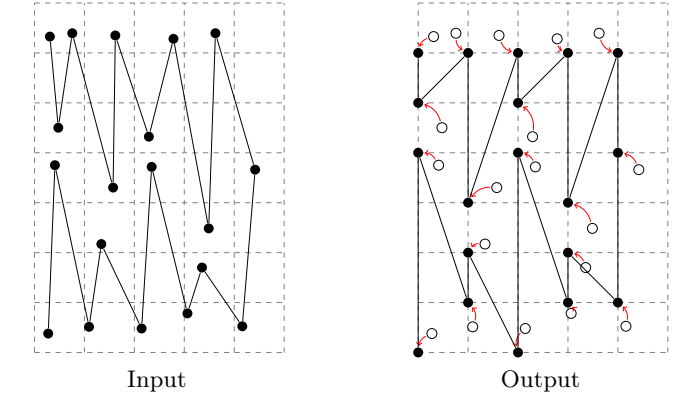


	Full model		Row generation	
Model size	rows	columns	rows	columns
Without presolve	*	*	24762	6482
With presolve	*	*	12355	3146
	first solution		optimal	optimal
Computation time	†	†	7.1 s	

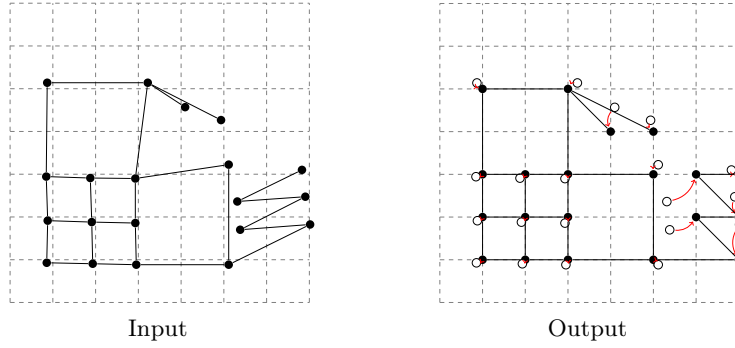
Fig. 5. Graph 4: $|V| = 26$, $|E| = 34$ 

	Full model		Row generation	
Model size	rows	columns	rows	columns
Without presolve	2603	916	2271	816
With presolve	2583	896	2245	682
	first solution		optimal	optimal
Computation time	0.5 s	4.8 s	20.2 s	

Fig. 6. Graph 5: $|V| = 7$, $|E| = 7$



	Full model		Row generation	
	rows	columns	rows	columns
Model size				
Without presolve	135386	35649	15741	4174
With presolve	74957	19591	9200	2402
	first solution	optimal	optimal	
Computation time	42.6 s	1105.6 s	21.2 s	

Fig. 7. Graph 6: $|V| = 19, |E| = 18$ 

	Full model		Row generation	
	rows	columns	rows	columns
Model size				
Without presolve	323441	82816	15161	4044
With presolve	323441	82816	15127	3894
	first solution	optimal	optimal	
Computation time	182.1 s	†	211.6 s	

Fig. 8. Graph 7: $|V| = 20, |E| = 25$